L49: Entry 4 of 36 File: USPT Jul 31, 2001

DOCUMENT-IDENTIFIER: US 6269436 B1
TITLE: Superscalar microprocessor configured to predict return addresses from a return stack storage

Detailed Description Text (1):
While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

Detailed Description Text (13):
Upon decode of a particular instruction, if a required operand is a register location, register address information is routed to reorder buffer 216 and register file 218 simultaneously. Those of skill in the art will appreciate that the x86 register file includes eight 32 bit real registers (i.e., typically referred to as EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP). Reorder buffer 216 contains temporary storage locations for results which change the contents of these registers to thereby allow out of order execution. A temporary storage location of reorder buffer 216 is reserved for each instruction which, upon decode, is determined to modify the contents of one of the real registers. Therefore, at various points during execution of a particular program, reorder buffer 216 may have one or more locations which contain the speculatively executed contents of a given register. If following decode of a given instruction it is determined that reorder buffer 216 has a previous location or locations assigned to a register used as an operand in the given instruction, the reorder buffer 216 forwards to the corresponding reservation station either: 1) the value in the most recently assigned location, or 2) a tag for the most recently assigned location if the value has not yet been produced by the functional unit that will eventually execute the previous instruction. If the reorder buffer has a location reserved for a given register, the operand value (or tag) is provided from reorder buffer 216 rather than from register file 218. If there is no location reserved for a required register in reorder buffer 216, the value is taken directly from register file 218. If the operand corresponds to a memory location, the operand value is provided to the reservation station unit through load/store unit 222.

Detailed Description Text (15):
Reservation station units 210A-210D are provided to temporarily store instruction information to be speculatively executed by the corresponding functional units 212A-212D. As stated previously, each reservation station unit 210A-210D may store instruction information for up to three pending instructions. Each of the four reservation stations 210A-210D contain locations to store bit-encoded execution instructions to be speculatively executed by the corresponding functional unit and the values of operands. If a particular operand is not available, a tag for that operand is provided from reorder buffer 216 and is stored within the corresponding reservation station until the result has been generated (i.e., by completion of the execution of a previous instruction). It is noted that when an instruction is executed by one of the functional units 212A-212D, the result of that instruction

is passed directly to any reservation station units 210A-210D that are waiting for that result at the same time the result is passed to update reorder buffer 216 (this technique is commonly referred to as "result forwarding"). Instructions are issued to functional units for execution after the values of any required operand (s) are made available. That is, if an operand associated with a pending instruction within one of the reservation station units 210A-210D has been tagged with a location of a previous result value within reorder buffer 216 which corresponds to an instruction which modifies the required operand, the instruction is not issued to the corresponding functional unit 212 until the operand result for the previous instruction has been obtained. Accordingly, the order in which instructions are executed may not be the same as the order of the original program instruction sequence. Reorder buffer 216 ensures that data coherency is maintained in situations where read-after-write dependencies occur.

Detailed Description Text (103):
This section describes the instruction cache organization, fetching mechanism, and pre-decode information. The Processor 500 instruction cache has basic features including the ICSTORE, ICTAGV, ICNXTBLK, ICCNTL, ICALIGN, ICFPC, and ICPRED. Highlights are: the pre-decode bits per byte of instructions are 3 bits, the branch prediction increases to 2 targets, 2 different types of branch prediction techniques (bimodal and global) are implemented, the X86 instructions align to 4 fixed length RISC-type instructions, and the pre-decode logic eliminates many serializtion conditions. Processor 500 executes the X86 instructions directly with a few instructions requiring two Rops, the BYTEQ is configured for fast scanning of instructions, and instructions are aligned to 4 decode units. The pre-decode data is separate in a block called ICPDAT, instead of inside the ICSTORE. The pre-fetch buffers are added to the ICSTORE to write instructions directly into the array, and the prefixes are not modified. All branches are detected during pre-decoding. Unconditional branches are taken during pre-decoding and aligning of instructions to the decode units. A return stack is implemented for CALL/RETURN instructions. Way prediction is implemented to read the current block and fetch the next block because the tag comparison and branch prediction do not resolve until the second cycle. The scanning for 4 instructions is done from ICPDAT before selected by tag comparison. Since the pre-decode data does not include the information for the 2-Rop instructions, the instructions must be partially decoded for the 2-rop during prioritizing and aligning of instructions to decode units. The early decoding includes decoding for unconditional branches, operand addresses, flags, displacement and immediate fields of the instruction. The CMASTER takes care of the replacement algorithm for the Icache and sends the way associative along with the data to the pre-fetch buffer. This section includes signal lists, timings and implementation issues for the Icache and all sub-blocks.

Detailed Description Text (268):
Processor 500 executes fast X86 instructions directly, no ROPs are needed. The pre-decode bits with each byte of instruction are 3 bits; start bit, end bit, and functional bit. All the externally fetched instructions will be latched into the Icache. This should not be a problem since the Icache is idle and waits for external instructions. Only single byte prefix of 0x66 and 0x0F is allowed for Processor 500's fast path, multiple prefixes including 0x67 is allowed for multi-prefix, all other prefixes will take an extra cycle in decoding or go to MROM. With these simple prefixes, the instruction bytes need not be modified. The linear valid bit is used for the whole cache-line of instructions, 16-byte. The replacement procedure is done by the CMASTER. Along with each line of instruction, the CMASTER tells the Icache which way to put in the data and tag. The start and end bits are sufficient to validate the instruction. If branching to the middle of the line or instructions wrapping to the next line, the start and end bits must be detected for each instruction or else the instruction must be pre-decoded again. The possible cases are branching to the opcode and skipping the prefix (punning of instruction) and part of the wrapping instruction is replaced in the Icache. The instructions must first pass through the pre-fetch buffers before sending to the ICPRED. The

ICPRED has only one input from the IB(127:0) for both the pre-fetched or cached instructions. The pre-decode information is written into the ICPDAT as the whole line is decoded. The output IB(127:0) is merged with the previous 8-byte to form a 24-byte line for the alignment unit to select and send to 4 decode units.

Detailed Description Text (271):
The ICSTORE on Processor 500 does not store the pre-decode data, as shown in FIG. 9. The ICSTORE consists of 32K bytes of instructions organized as 8 sets of 128 rows by 256 columns. The array set in this documentation has its own decoder. The decoder is in the center of the set. Each of the sets consist of 2-byte of instructions. The 8-way associative muxing from the 8 TAG-HITs is performed before the data is routed to the ICALIGN block. With this arrangement, the input/output to each set is 16-bit buses. The muxing information relating to which byte is going to which decode unit should also be decoded; this topic will be discussed in detail below in the ICALIGN block section. For optimal performance the layout of the column should be 64 RAM cells, pre-charge, 64 RAM cells, write buffer and senamp. The row decoder should be in the middle of the array to drive 128 column each way. Basically, the pre-charge and the row decoder should be crossed in the middle of the array. The self-time column is used to generate internal clock for each set of the array. Pre-charge is gated by ICLK. The instruction is valid by the end of ICLK, the data muxes by the TAG-HIT should be gated by ICLK to be valid for the second ICLK. The two-entry pre-fetch buffers are implemented inside the array with data can be written from either entry. The output IB bus is taken from either the array or the pre-fetch buffer.

Detailed Description Text (322):
The ICPDAT includes of 32K of 3-bit pre-decode data organized as 8 sets of 64 rows by 192 columns. Each of the sets consists of two 3-bit pre-decode data. The pre-decode data is decoded into byte-shifting information which ise used by the ICALIGN block. The 8-way associative muxing from the 8 TAGHITs is performed before the byte-shifting data is routed to the ICALIGN block. In order for the instructions to get to the Idecode in middle of the second ICLK, the decode logic for the byte-shifting should be less than 7 gates. The byte-shifting logic has been done. Because of this byte-shifting logic, the array for ICPDAT is 64 rows instead of 128 rows for the ICSTORE array. For optimal performance the layout of the column should be 32 RAM cells, pre-charge, 32 RAM cells, write buffer and senseamp. The row decoder should be in the middle of the array to drive 96 column each way. Basically, the pre-charge and the row decoder should be crossed in the middle of the array. The self-time column is used to generate internal clock for each set of the array. Pre-charge is gated by ICLK. The byte-shifting data muxed by the TAGHIT should be gated by ICLK to be valid for the second ICLK. The output of the array should include logic to feedback the previous pre-decode data for breaking up of the line for second cycle access.

Detailed Description Text (386):
The status bits need to be dual-port to read and write in the same clock cycle. The ICTAGV is organized as two sets of 64 rows by 224 columns and two sets of 64 rows by 128 columns. Each of the first two sets consists of 7-bit tag address, and each of the last two sets consists of 3-bit tag address and the SU or LV bit. The two status bits are dual port RAM cells. The SU uses the delayed PC to write, and the LV bit has the snooping index from CMASTER. The ICTAGV uses 64 rows for dual-port RAM and quick reading of tag address. For optimal performance the layout of the column should be 32 RAM cells, pre-charge, 32 RAM cells, write buffer and senamp. The row decoder should be in the middle of the array to drive 112 or 96 columns each way. Basically, the pre-charge and the row decoder should be crossed in the middle of the array. The row decoder for the dual port RAM should be build at one end of the array. The self-time column is used to generate internal clock for each set of the array. Pre-charge is gated by ICLK. The status bits muxed by the TAGHIT should be gated by ICLK to be valid for the second ICLK. The above layout is to ensure the minimum routing for the TAGHIT, and is shown in FIG. 12.

Detailed Description Text (426):
This branch prediction is an independent branch predictor, not a part of the Icache. There are many different types of global branch prediction; Processor 500 uses the global branch prediction which has the highest branch correct prediction ratio. The prediction entries are indexed by an exclusive OR of the PC and the branch shift register, which is referred to as global sharing branch prediction. This global branch prediction has the branch correct prediction at 89.24%; the prediction improves as more branch history bits are used in the prediction. A single shift register records the direction taken/non-taken by the most recent n conditional branches. Since the branch history is global to all branches, global branch prediction takes advantage of two types of patterns, the direction taken by the current branch may depend strongly on the other recent branches, and the duplicating the behavior of local branch prediction (patterns of branches in loops). For Processor 500, since the highest priority of the two branch targets is taken branch, the global shift register includes both the conditional and unconditional branches. In a few cases, the non-taken conditional branches may not include in the global shift register. To match the number of entries in the Icache, the global branch prediction needs to have 2048 entries with 2 targets per entry. It is organized with 256 rows of 8-way associative. Eight bits are used to index the branch prediction table. The PC uses bit 11:4 for indexing the branch prediction table.

Detailed Description Text (483):
The ICNXTBLK is organized as 4 sets of 64 rows by 256 columns, 2 sets of 64 rows by 196 columns including some dual-ported columns, 1 set of 64 rows by 128 dual-ported column, 1 set of 64 rows by 96 dual-ported columns, 1 set of 64 rows by 64 dual-ported columns, and 1 set of global counter array. Each of the first two sets consist of 2.times.4 bits of successor index. The next two sets consists of 2.times.4 bits of successor index and 2.times.4 bits of the byte position. The next two sets consists of 2.times.2 bits bimodal counter, 2.times.2 bits predictor counter, and 2.times.3 bits 8-way associative, the least significant bits of the counters are dual-port. The next set consists of 4 bits of way branch byte pointer. The last two sets consist of the 3 bits way-prediction and 2 bits target selection which are dual-ported RAM cells. The least significant bits of the counters are dual ported to be updated on every cycle. To minimize routing and implementation of the branch holding register, the same associated bits of the two branch targets should be laid out in two sets opposite each other. The branch successor index is selected by the way and target prediction to access the ICACHE in next clock cycle. Because of this speed-path in way prediction for reading the Icache in the next cycle, the array for ICNXTBLK is 64 rows instead of 128 rows as for the ICSTORE array. For optimal performance the layout of the column should be 32 RAM cells, pre-charge, 32 RAM cells, write buffer and senamp. The row decoder should be in the middle of the array to drive 96 or 112 column each way. Basically, the pre-charge and the row decoder should be crossed in the middle of the array. The self-time column is used to generate internal clock for each set of the array. Pre-charge is gated by ICLK. The ICNXTBLK has two different outputs; the first output in the first cycle is based on the way-prediction, and the second output in the second cycle is based on TAGHIT. If the two outputs do not select the same set or are not both non-taken, the reading of instruction in the second cycle will be invalidated, creating a bubble in the pipeline. The second output should be gated with TAGHIT and ICLK to be valid in the second cycle.

Detailed Description Text (588):
Build-In Self-Test counter.

Detailed Description Text (601):
Build-In Self-Test counter.

Detailed Description Text (613):

Build-In Self-Test counter.

Detailed Description Text (626):
Build-In Self-Test counter.

Detailed Description Text (717):
Decode for Conditional JUMP is needed to keep it in the ICNXTBLK and for global branch prediction. If the target address of the conditional JUMP (and LOOP) can be calculated and the branch jumps backward, then it is predicted taken. Since backward branch is mostly taken and the adder is available to calculate the target address, the conditional branch should be predicted taken. The taken branches have the higher priority to occupy the branch targets. Conditional branch has higher priority to occupy two branch targets in the ICNXTBLK than CALL or Unconditional JUMP with 8-bit displacement linear address. The decoding for all branches are needed to set the global shift register. If the target address is a simple calculation, the decode units should calculate the target address and jump to new block.

Detailed Description Text (788):
A lock prefix is only allowed on certain instructions. When not applicable, lock prefixes cause an illegal opcode exception. Some instructions (i.e. EXCH) cause locked accesses by default without the lock prefix. Multiple lock prefixes do the same as single lock prefix.

Detailed Description Text (791):
Besides the SIB instruction which requires two dispatch positions, few other instructions must go through decoding. The same opcode are sent to two decode units with an encoded field for indication of the first or second dispatch position. The list of 2-dispatch position instructions are: PUSH, POP, RETURN, CALL, MUL, IMUL, LOOP, JCXZ, JECXZ, and LEAVE. The above instructions have either two destination registers, or two load/store operations, or more than two input operands (not counting the immediate and displacement). The above instructions with SIB should go to MROM. PUSH from memory and POP to memory instructions can become register if the MOD field is 11. In this case, the PUSH from memory should take only one dispatch position, and the POP to memory should take 2 dispatch positions instead of going to MROM.

Detailed Description Text (806):
A constant field is generated for the ESP calculation. Depending on the address size, a constant of 2 or 4 is needed for the above instructions. For JCXZ and JECXZ instructions, a constant of 0 is needed for the zero detection. For LOOP instructions, a constant of 1 is needed for decrementing of the counter.

Detailed Description Text (977):
It is important to implement testability features into Processor 500 to reduce test time, burn-in time, and increase fault coverage. The Build-In-Self-Test (BIST) for the arrays and Auto-Test-Pattern-Generation for the random logic are included.

Detailed Description Text (1002):
The Build-In Self-Test (BIST) uses the Test Application and Error Compression (TAEC) cells for reading and writing the arrays. Each TAEC cell includes an input shift register for the test pattern and one output shift register for the result. All TAEC cells are connected to form a serial shift path.

Detailed Description Text (1004):
The ATPG is implemented to test non-array blocks in the Icache. The purpose is to be able to reach any node in the logic. The feedback (loop) paths (state machines) must be broken with scan latches. There is a software algorithm to insert the scan latch into the logic. The control blocks in Icache should include the ATPG inputs and outputs for the software to use.

Detailed Description Text (1025):
Since the clock cycle is short, reading of the cache would take the whole clock to get data. The clock is single phase, and the array generates its own self time clock. The self-time clock uses the same cache column self-time line. As the line pre-charges to a high level, the pre-charge is disabled and the array access is enabled. As the line discharges, the row driver and senamp are disabled. In one embodiment, the pre-charge takes 1.7 ns and the current timing for TAGHIT from the self-time clock with 64 rows is 2.8 ns or a total time of 4.5 ns from rising edge of ICLK. The reading data is 2.0 ns from the self-time clock with 64 rows or 0.8 ns before the rising edge of ICLK. The ICSTORE can be built with larger arrays, 128 rows by 256 columns, reading instructions would take all of 4.5 ns ICLK in this case. Other arrays, ICTAGV, ICPRED, and ICNXTBLK, are 64 rows. The align logic in the ICPDAT takes 6-7 gates, and the shifting of X86 instruction bytes to the decode unit can be done by the middle of the second ICLK. The Processor 500 instructions should allow the decode units at least 2.5 ns in the second ICLK for calculation of the linear address.

Detailed Description Text (1056):
Decoding of other instruction fields such as lock, segment register controls, special register, and floating point unit.

Detailed Description Text (1108):
IDxLOCK--Output indicates the lock prefix is set for this instruction for serialization.

Detailed Description Text (1116):
INSLSxB(5:0)--Output from decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (1178):
INSLSxB(5:0)--Output from decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (1213):
IDPREF(5:0)--Output from 2-cycle prefix decode to decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (2592):
INSLSxB(5:0)--Input from decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (2597):
EXCHGSYNC--Output indicates exchange instruction resynchronization to Icache. This occurs when an exchange with a masked underflow is retired. It is a special resynchronize exchange with alternate entry point.

Detailed Description Text (2634):
WBEXCHG--Output to register file indicates the exchange instruction being retired. It causes the permanent remapping register to be updated from the write-back bus.

Detailed Description Text (2640):
The FIROB interfaces with the decode units for dispatching instructions, with the functional units and LSSEC for results, and with the register file for retiring instructions. The FIROB updates the special registers correctly with each retiring instructions, handles trap/interrupt gracefully, and re-synchronizes the pipeline after branch mis-prediction, self-modifying code, or changing the code segment register.

Detailed Description Text (2660):
For internal exceptions from the functional units, LSSEC, and SRB, the exception entry in the FIROB will be retired in order. Similar to the branch mis-prediction, the pipe and fetching should stop on an exception indication. When all entries before the exception entry have completed and retired, the exception procedure is initiated. All entries in the FIROB, the functional units, and LSSEC will be purged. The exception routine will be fetched. The FIROB is responsible to generate the entry point to the MROM exception routine or new PC to the Icache. No state is updated when a trap is taken. The processor fetches from an appropriate entry point and allows the microcode to perform the necessary state modifications. It is up to the microcode to save the current EIP on the stack before the user's trap handler is called.

Detailed Description Text (2678):
INSLSxB(5:0)--Input from decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (2715):
Another condition to re-fetch the instructions is re-synchronization. There is a possibility of executing a wrong instruction from self-modifying code and updating the code segment register. The stream of instructions must be re-fetched. From external snooping which cause an invalidation of a line in the instruction cache or an internal store instruction which can change a line in the instruction cache, the FIROB is re-synchronized at this point. As soon as the write to code-segment register is detected, following instructions are re-fetched upon completion of the code segment register write.

Detailed Description Text (2764):
On external interrupts, the entry point is generated locally by the FIROB at the time the redirection is initiated. The maskable interrupt is level sensitive while the NMI is edge sensitive. FLUSH and INIT are also treated as edge sensitive asynchronous interrupts, similar to NMI. If the NMI is taken, it cannot be taken again before an IRET is executed. The microcode maintains a series of global flags that are inspected and modified by many of the trap handler entry points, and the IRET instruction. It is also the responsibility of the microcode to detect the NMI and delay the NMI until after executing of the IRET. The MROM allows only one level of NMI. Many other aspects of nested trap control (double fault, shutdown, etc.) will be handled with this microcode mechanism. There is no hardware support for this. When an enabled trap condition arises, the FIROB takes it at the next available window.

Detailed Description Text (2802):
EXCHGSYNC--Output indicates exchange instruction resynchronization to Icache. This occurs when an exchange with a masked underflow is retired. It is a special resynchronize exchange with alternate entry point.

Detailed Description Text (2879):
INSLSxB(5:0)--Input from decode units indicates the prefix values. bit 5--data size, bit 4--address size, bit 3--lock, bit 2:0--segment registers.

Detailed Description Text (2887):
ROBLSYNC--LOAD/STORE RESYNC--1-bit--Set when functional units return valid results with resync status. The load/store hits in the Icache for self-modifying code. The next instruction should be re-fetched from the Icache.

Detailed Description Text (3044):
A register file timing diagram is illustrated in FIG. 37. Instead of the normal method of a write to the register file followed by a read from the register file, Processor 500 does a read first followed by a write. The early read allows sufficient time for multiplexing between the register value (the one actually read)

and forwarding from the incoming writes. The end of the cycle is needed to drive the read value over to the operand steering unit. A self timing circuit is used to provide sufficient delay for the write and read decoding logic to complete before the read and write actually take place. Both the read and write decoding sections start decoding immediately after the ALAT's latch in the read and write pointer busses. There are sixteen 5 bit comparators which detect if forwarding is needed from a writeback port to a read port. The forwarding will bypass the delay through the register latch and help allow the read to complete within the cycle. The Read outputs from the register file will drive on a dedicated bus over to the operand steering unit. For maintaining fast logic, 3 input nand gates are used in the decode section. The 4 input nand gate is a large decrease in speed, and the 5 bit pointer bus along with an enable signal fit a two 3 input nand gate structure. There is not any reset logic for the register array.

Detailed Description Text (3091):
If the CMASTER reports that the line is in the ICACHE (self-modifying code) and if the access happens to be a store, the LSSEC sends a "store-with-resync" status to the ROB to flush out the instructions after the store.

Detailed Description Text (3199):
During prefetch of a cacheable line, the store buffer entries need to be snooped make sure that a store in the store buffer is not to the same prefetched line. If it is (self-modifying code case), the prefetch stalls till the store is written.

Detailed Description Text (3262):
3. Protocol for handling locked accesses

Detailed Description Text (3491):
D is the dirty bit that indicates that the line has been previously modified. This information is used during a store by the LSSEC when the TLB is accessed to determine whether the corresponding dirty bit in the page table entry is correctly set. If the dirty bit in the page table entry is not set then an exception must occur to write the dirty bit in the external page table entries so that the page gets written back to external memory.

Detailed Description Text (3535):
The dcache is involved when an inquire cycle hits a modified line since a writeback cycle is issued to update the modified line in external memory. The dcache is also involved during snoop invalidations.

Detailed Description Text (3723):
It is further noted that aspects regarding array circuitry may be found in the co-pending, commonly assigned patent application entitled "High Performance Ram Array Circuit Employing Self-Time Clock Generator for Enabling Array Access", Ser. No. 08/473,103 filed Jun. 7, 1995 by Tran, now U.S. Pat. No. 5,619,464. The disclosure of this patent application is incorporated herein by reference in its entirety.

Detailed Description Text (3726):
Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

Detailed Description Paragraph Table (13):
Prefix Encoded Prefix 0000 0000 No prefix 0x0F 1xxx xxxx 2-byte escape 0x66 x1xx xxxx operand size override, 16 or 32-bit data 0x67 xx1x xxxx address size override, 16 or 32-bit addr 0xF0 xxx1 xxxx LOCK prefix 0x2E xxxx 1000 CS segment override 0x36 xxxx 1001 SS segment override 0x3E xxxx 1010 DS segment override 0x26 xxxx 1100 ES segment override 0x64 xxxx 1101 FS segment override 0x65 xxxx 1110 GS segment override

Detailed Description Paragraph Table (14):
Opcode 2-dispatch positions 0000000000 Single instruction 0xxxxxxxxx First rop of the 2-dispatch positions instruction 1xxxxxxxxx Second rop of the 2-dispatch positions instruction 0xFF /6 x1xxxxxxxx PUSH from memory 0x58 xx1xxxxxxx POP into register 0x1F POP into DS 0x07 POP into ES 0x17 POP into SS 0x0F A1 POP into FS 0x0F A9 POP into GS 0xF7 /4 xxx1xxxxxx MUL instruction - word/dword 0xF7 /5 xxx1xxxxxx IMUL instruction - word/dword 0xC9 xxxx1xxxxx LEAVE 0xE0 xxxxx1xxxx LOOP with ZF=0 0xE1 LOOP with ZF=1 0xE2 LOOP 0xE3 xxxxxx1xxx JCXZ & JECXZ 0xE8 xxxxxxx1xx CALL near, displacement relative 0xFF /2 CALL near, register indirect relative 0xC3 xxxxxxxx1x RETURN near 0xC2 RETURN near, immediate FUNC bit set xxxxxxxxx1 SIB-byte instruction

Detailed Description Paragraph Table (26):
dispatch positions 1st pos 2nd pos regular operation OP / - ; regular op w/ imm OP / - ; Load-op LO / - ; OP-store OS / - ; Load-op-store LOS / - ; SIB w/ Load-op SIB / LO ; SIB w/ OP-store SIB / OS ; SIB w/ L-O-S SIB / LOS ; divide opcode DIV / - ; FPU linear adr FLA / - ; Jcc BRN / - ; JMP nr disp rel BRN / - ; JMP nr reg indr BRN / - ; JMP nr mem indr LBR / - ; PUSH reg PU / - ; PUSH mem LO / PU ; PUSH mem w/SIB uCode PUSHF Pu / - ; using MOVF PUSHF w/OF fwd V2 / PU ; using MOVF and MOVOF CALL nr disp rel PU / BRN ; CALL nr reg indr PU / BRN ; pos 2 adds indr_reg w/ zero instead of EIP+rel CALL nr mem indr PU / LBR ; POP mem uCode POP mem w/SIB uCode POPF uCode ; this goes to uCode since IOPL can change RET LBR / OP ; RET imm LBR / V12 ; XCHG uCode XADD uCode LEAVE POP / LO ; LOOP OP / BRN ; LOOPcond uCode JCXZ OP / BRN ; MUL 1 disp pos OP / - ; MUL 2 disp pos OP / OP ; 2nd pos is a NOP

Detailed Description Paragraph Table (27):
regular operation OP / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A B -- -- v0: none {0} v1: A B operation {F} same but with immediate data regular operation OP / - ; OP_STR block moves imm to Bop dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A imm -- -- v0: none {0} v1: A imm operation {F} Load-op LO / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> B disp -- v0: B disp Bop+disp {Lw} & Btag<=Dtag v1: LSRES operation {F} OP-store OS / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A B disp -- v0: A disp Aop+disp {L} v1: B operation {M} Load-op-store LOS / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A B -- -- v0: A disp Aop+disp {Lw} & Atag<=Dtag v1: LSRES B operation {M} SIB w/ Load-op SIB / LO ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> base index -- -- v0: base index base+scaled {F} (ind) v1: none {0} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> B disp -- v0: fwd disp fwd+disp {Lw} & Btag<=Dtag v1: LSRES operation {F} SIB w/ OP-store SIB / OS ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> base index -- -- v0: base index base+scaled {F} (ind) v1: none {0} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> A B disp -- v0: fwd disp fwd+disp {L} v1: B operation {M} SIB w/ L-O-S SIB / LOS ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> base index -- -- v0: base index base+scaled {F} (ind) v1: none {0} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> A B disp -- v0: fwd disp fwd+disp {Lw} & Atag<=Dtag v1: LSRES B operation {M} divide opcode DIV / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A B -- -- v0: A B ADD A,B {F} v1: A B SUB A,B {F} FPU linear address FLA / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> B disp const v0: B disp Bop+disp {I} & Btag<=Dtag v1: fwd const fwd+const {L} Jcc BRN / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> EIP pre_adr rel -- v0: none {0} v1: EIP B rel EIP+rel {B} next cycle flag logic compares EIP+rel & pre_adr for predicted taken JMP nr disp rel BRN / - ; same as conditional except always taken dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> EIP pre_adr rel -- v0: none {0} v1: EIP B rel EIP+rel {B} next cycle flag logic compares EIP+rel & pre_adr for predicted taken JMP nr reg indr BRN / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> reg pre_adr -- 0 v0: none {0} v1: reg 0 reg+0 {B} next cycle flag logic compares reg+0 & pre_adr for predicted taken JMP nr mem indr LBR / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> Aop pre_adr disp 0 v0: Aop disp Aop+disp {Lw} & Atag<=Dtag v1: LSRES 0 LSRES+0 {B} next cycle flag logic compares LSRES+0 & pre_adr for

predicted taken PUSH reg PU / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP B -- const v0: ESP const ESP-const {L,F} v1: B operation {M} PUSH mem LO / PU ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> B disp -- v0: B disp Bop+disp {Lw} & Btag<=Dtag v1: LSRES operation {F} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> ESP fwd -- -- v0: ESP const ESP-const {L,F} v1: fwd operation {M} PUSHF PU / - ; using MOVF dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP flg -- const v0: ESP const ESP-const {L,F} v1: flg MOVF {M} the MOVF on v1 combines system flags on Bop with CF & XF PUSHF w/OF fwd OP / Pu ; using MOVF and MOVOF dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> -- flg -- -- v0: none {0} v1: flg MOVF {F} the MOVF on v1 combines system flags on Bop with CF & XF dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> ESP fwd -- const v0: ESP const ESP-const {L,F} v1: fwd MOVOF {M} the MOVOF on v1 overwrites the OF bit position of Bop CALL nr disp rel PU / BRN ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP EIP -- const v0: ESP const ESP-const {L,F} v1: EIP operation {M} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> EIP pre_adr disp -- v0: none {0} v1: EIP disp -- EIP+disp {B} next cycle flag logic compares EIP+disp & pre_adr for predicted taken CALL nr reg indr PU / BRN ; pos 2 adds indr_reg w/ zero instead of EIP+rel dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP EIP -- const v0: ESP const ESP-const {L,F} v1: EIP operation {M} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> reg pre_adr 0 -- v0: none {0} v1: reg 0 -- reg+0 {B} next cycle flag logic compares reg+0 & pre_adr for predicted taken CALL nr mem indr PU / LBR ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP EIP -- const v0: ESP const ESP-const {L,F} v1: EIP operation {M} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> A pre_adr disp -- v0: A disp Aop+disp {Lw} & Atag<=Dtag v1: LSRES LSRES+0 {B} next cycle flag logic compares LSRES+0 & pre_adr for predicted taken POP reg LO / POP ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP 0 -- v0: ESP 0 ESP+0 {Lw} & Btag<=Dtag v1: LSRES operation {F} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> ESP -- -- const v0: ESP const ESP+const {F} v1: none {0} RET LBR / OP ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP pre_adr -- 0 v0: ESP 0 ESP+0 {Lw} & Atag<=Dtag v1: LSRES 0 LSRES+0 {B} next cycle flag logic compares LSRES+0 & pre_adr for predicted taken dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> ESP -- -- const v0: ESP const ESP+const {F} v1: none {0} RET imm LBR / V12 ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ESP pre_adr 0 -- v0: ESP 0 ESP+0 {Lw} & Atag<=Dtag v1: LSRES 0 LSRES+0 {B} next cycle flag logic compares LSRES+0 & pre_adr for predicted taken dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> ESP imm const v0: ESP const ESP+const {I} v1: fwd imm fwd+imm {F} LEAVE V12 / LO ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> EBP -- const v0: const EBP+const {I} v1: fwd R<-fwd {F} this one is used so another protocol is not needed and dependency checking is easy; otherwise, use a version of the POP protocol with Bop+const. FIROB latches the output as the new ESP value. dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> EBP 0 -- v0: EBP 0 EBP+0 {Lw} & Btag<=Dtag v1: LSRES operation {F} FIROB latches v1 result as the new EBP value LOOP Op / BRN ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ECX 1 -- -- v0: none {0} v1: ECX 1 operation {F} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> EIP pre_adr disp -- v0: none {0} v1: EIP disp -- EIP+disp {B} next cycle flag logic compares EIP+disp & pre_adr for predicted taken JCXZ OP / BRN ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> ECX 0 -- -- v0: none {0} v1: ECX 0 operation {F} dsptch pos 2: Aop Bop Disp Const operation Latch inputs-> EIP pre_adr disp -- v0: none {0} v1: EIP disp -- EIP+disp {B} next cycle flag logic compares EIP+disp & pre_adr for predicted taken MUL 1 disp pos OP / - ; dsptch pos 1: Aop Bop Disp Const operation Latch inputs-> A B -- -- v0: none {0} resv_sta receives grant and sends source on SAOPND & SBOPND; FNCU is idle resv_sta starts counting 3 cycles and issues v1 to pass MUL result v1: pass MUL {F} MUL 2 disp pos OP / Op ; 2nd pos is a NOP

Detailed Description Paragraph Table (52):
x86 Assembly Fast Path Opcode seen by FNCU AAA uCode AAD uCode AAM uCode AAS uCode ADC fast ADC ADD fast ADD AND fast AND ARPL uCode BOUND uCode BSF fast BSF (may become uCode eight bits at with 16 bit BSF) a time BSR fast BSR (may become uCode

h e b b g e e e f c e b e ge

eight bits at with 16 bit BSR) a time BSWAP uCode BT fast BT BTC fast BTC BTR fast BTR BTS fast BTS CALL fast near indirect or near relative else uCode CBW fast SIGNXA CWDE fast SIGNXA CLC fast use flag equations CLD fast use flag equations CLI uCode CLTS uCode CMC fast use flag equations CMP fast SUB (no result writeback by FIROB) CMPS uCode CMPSB uCode CMPSW uCode CMPSD uCode CMPXCHG uCode CMPXCHG8B uCode CPUID uCode CWD uCode CWQ uCode DDA uCode DAS uCode DEC fast SUB wo/ CF save DIV uCode ENTER uCode HLT uCode IDIV uCode IMUL fast IMUL, some are 2 dispatch pos IN uCode INC fast ADD wo/ CF save INS uCode INSB uCode INSW uCode INSD uCode INT uCode INTO uCode INVD uCode INVLPG uCode IRET uCode IRETD uCode Jcc fast JMPCC JCXZ fast two dispatch positions JECXZ fast two dispatch positions JMP fast near and indirect else uCode LAHF fast LAHF LAR uCode LDS uCode LES uCode LFS uCode LGS uCode LSS uCode LEA fast LEAB LEAVE fast two dispatch position LGDT uCode LIDT uCode LLDT uCode LMSW uCode LODS uCode LODSB uCode LODSW uCode LODSD uCode LOOP fast two dispatch positions LOOPcond uCode LSL uCode LTR uCode MOV fast MOVZX MOVCC fast MOVCC MOV CR uCode MOV DR uCode MOVS uCode MCVSB uCode MOVSW uCode MOVSD uCode MOVSX fast MOVSX MOVZX fast MOVZX MUL fast MUL; some are two dispatch pos NEG fast SUB NOP not sent to FNCU NOT fast SUB OR fast OR OUT uCode OUTS uCode OUTSB uCode OUTSW uCode OUTSD uCode POP fast two dispatch positions POPA uCode POPAD uCode POPF uCode-may change IOPL POPFD uCode-may change IOPL PUSH fast some are two dispatch pos PUSHA uCode PUSHAD uCode PUSHF fast MOVF PUSHFD fast MOVF RCL uCode using RCL_1 RCR uCode using RCR_1 ROL fast ROL ROR fast ROR RDMSR uCode REP uCode REPE uCode REPZ uCode REPNE uCode REPNZ uCode RET fast two dispatch positions RSM uCode SAHF fast SAHF SAL fast SHL SAR fast SAR SHL fast SHL SHR fast SHR SBB fast SBB SCAS uCode using DFADD SCASB uCode using DFADD SCASW uCode using DFADD SCASD uCode using DFADD SETcc fast SETCC SGDT uCode SIDT uCode SHLD uCode SHRD uCode SLDT uCode SMSW uCode STC fast ZEROXA & flags=BCD_FS STD fast FIROB STI uCode STOS uCode STOSB uCode STOSW uCode STOSD uCode STR uCode SUB fast SUB TEST fast AND (no result writeback by FIROB) VERR uCode VERW uCode WBINVD uCode WRMSR uCode XADD uCode XCHG uCode XLAT fast MOV XLATB fast MOV XOR fast XOR

Detailed Description Paragraph Table (93):
LS_FAKE_LOAD Output to CMASTER ICLK5 LS_FAKE_LOCK Output to CMASTER ICLK5 LSINDXCT Output to DCACHE ICLK14

Detailed Description Paragraph Table (96):
LS_LOCK Output of CMASTER ICLK5

Detailed Description Paragraph Table (97):
LS_LOCK Output of CMASTER ICLK5